AD-A044 634 HONEYWELL INFORMATION SYSTEMS INC MCLEAN VA F/G 9/2
WWMCCS ADP STANDARD CLEAR MEMORY/MAGNETIC STORAGE UTILITY STRUC--ETC(U)
JUN 77 J KARAS, B HICKEY, D MACKELLAR DCA100-73-C-0055
UNCLASSIFIED CCTC-WAD-TR-121-77 NL

| OF |
ADA044634

END
DATE FILMED
10-77
DDC

TR-121-77

(5)

Defense Communications Agency

Command and Control Technical Center

Contract DCA 100-73-C-0055

WWMCCS

Honeywell

WWMCCS ADP STANDARD
CLEAR MEMORY/MAGNETIC STORAGE UTILITY
STRUCTURED PROGRAMMING EVALUATION
TECHNICAL REPORT
TASK 621, SUBTASK 2
June 20, 1977

(16) CCTC-WAD

(17) TR-121-77

Honeywell

CCTC TASKING STATEMENT 621
SUBTASK 2

(9) Final rpt.
29 Mar — 21 Jun 77

SUBJECT: (6) WWMCCS ADP STANDARD CLEAR MEMORY/MAGNETIC STORAGE UTILITY STRUCTURED PROGRAMMING EVALUATION TECHNICAL REPORT.

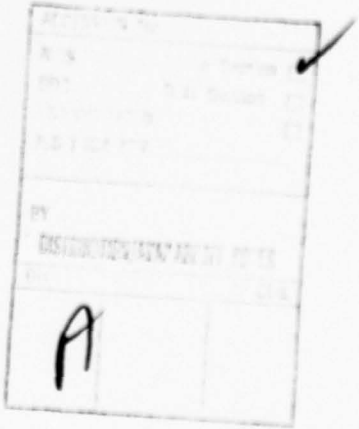
PREPARED FOR: Defense Communications Agency
Command and Control Technical Center
WWMCCS ADP Directorate

CONTRACT NUMBER: (15) DCA100-73-C-0055 (12) 41p.

CONTRACTOR: Honeywell Information Systems, Inc.

PREPARED BY: (10) Jeff Karas
Bob Hickey
Don MacKellar

DATE: (11) 21 June 20, 1977



408 521

LB

Table of Contents

INTRODUCTION

The requirement for an organized approach to the development of both operating system and application software is increasing. This requirement is felt more intensely in the development of large, complex systems which will be maintained over a long period of time and/or subjected to extensive modification. However, an organized approach can benefit the development of small scale systems also.

A conservative estimate of the total annual cost for software development is approximately 20 billion dollars [4]. Considering this, plus the fact that the demand for new software is growing at a rate of approximately 21-23% per year versus a growth rate in the software labor force of only about 11.5-17% [4], it becomes obvious that something must be done to make the software development process more efficient. Specifically, emphasis must be placed on 1) increasing the productivity of the labor force, 2) increasing the reliability of the software that is developed, and 3) reducing the percentage of total development resources currently being spent on software maintenance and modification.

It is important to emphasize the inclusion of the maintenance and modification functions in the definition of software development. An advertised benefit of many currently popular software development techniques is that they reduce maintenance costs and headaches, a very intriguing benefit since maintenance and modification costs may amount to 75% of the total software dollar by 1985 [4].

Structured Programming technology is currently emerging as a possible solution to this requirement for an organized approach to software development. Using data gathered during the structured development of a large scale software system, the various aspects of Structured Programming will be discussed and evaluated. The purpose of this paper is not to supply any essentially new dimensions to the theory, but rather to evaluate the impact of its use on the software development process. The paper consists of five major sections: (1) Structured Programming Methodology, which describes and evaluates Structured Programming techniques; (2) Management, which addresses management functions in a structured environment; (3) Benefits/Costs, which discusses the benefits and costs of the structured approach to software development; (4) Conclusions, which summarizes the evaluation of the gathered data; and (5) Recommendations, which presents recommendations on how and when to apply the various Structured Programming techniques.

The application focused upon is the Clear Memory Utility (CMU), an automated facility to purge the World Wide Military Command and Control System (WWMCCS) standard ADP system, the Honeywell Series 6000, of magnetically stored data between periods of secure processing. The Clear Memory Utility was developed by Honeywell Information Systems, Federal Systems Operations (FSO), under contract to the Defense Communications Agency (DCA), Command and Control Technical Center, WWMCCS ADP Directorate (CCTC/WAD).

As a 5.75 man-year Structured Programming project utilizing the talents of six people over a 13-month period, the development task was, for two reasons, more complex than its title, Clear Memory Utility, indicates.

First, because of stringent security requirements, the program is necessarily a totally independent system, with no support from GCOS, the standard Series 6000 operating system. From bootstrap of the program at the operator's console through final program termination, CMU supplies its own logic for loading, configuration verification and modification, input/output services, fault handling, clearing and self-erasure.

Secondly, it was required that the developed system be verifiable as to correctness. Theoretically, Structured Programming methodologies coupled with specification and implementation languages which submit themselves to mathematical proofs of correctness would supply a universe to meet this requirement.

It was because of the verifiability requirement of the contract, as well as to test state-of-the-art productivity enhancement claims alleged by the Structured Programming literature, that Honeywell analysts developed the Clear Memory Utility in the Structured Programming environment described on the following pages.

1.0 STRUCTURED PROGRAMMING METHODOLOGY

A brief discussion of the principles of Structured Programming will serve to identify the major elements as they relate to the Clear Memory Utility.

The concept of Structured Programming emerged in 1968 when Professor E. W. Dijkstra's cornerstone letter to the ACM entitled "GOTO-Statement Considered Harmful" [9] appeared. Other authors followed, each adding a new dimension to the Structured Programming concept. C. Böhm and G. Jacopini suggested that it was possible to write programs using only three logical structures: simple sequence, selection (if-then-else), and repetition (do-while or do-until) [5]. Dr. Harlan Mills of IBM relaxed the concept of no goto statements to allow forward goto statements but insisted on only one entry and one exit point for each program module [14]. Mills, along with F. T. Baker of IBM, proved the viability of combining the Chief Programmer Team and the Development Support Library with earlier concepts of Structured Programming [3].

Slowly, well-defined Structured Programming methodologies evolved. While there are current disagreements on some issues, it is generally recognized that Structured Programming technology includes the following major elements:

A. Top-down design, construction, implementation, and maintenance

B. Limited and strictly observed technical procedures such as:

   1. Strict functional modularity of code
   2. Limited program structures and conventions
   3. Documentation standards

C. Development Support Library

D. Chief Programmer Team

The application of these elements of Structured Programming technology to the development of the Clear Memory Utility will be discussed in the following sections.

1.1 Top-Down Design

A basic tenet of Structured Programming theory is a top-down approach to design, construction and

implementation. Each phase of system development is executed in a top-to-bottom and beginning-to-end order. The problem is decomposed hierarchically into a series of logical steps with as few decisions made at each step as possible. Theoretically, this hierarchical design will force the organization of a problem solution along natural algorithmic boundaries and each resultant module will perform a single, specific, well-defined function.

The first phase of top-down design used in the Clear Memory Utility involved the construction of levels-of-abstraction phrases. Continually asking the question what to do to solve a problem, the designers began at the most general level and defined a set of phrases to describe the problem. These first-level phrases were then broken down using the same what type approach and the resulting phrases were also broken down. For example, the phrase for a first-level module (DOTASK) was: DOTASK initializes hardware/software communication areas, sets up and verifies the installation hardware configuration, boots the Front-End Network Processors (FNPs), performs erase procedures, and terminates. Each of these five functions was then defined as a second-level module (DOINIT, DOCFIG, DOBOOT, DOCLER, and DOEXIT). The level-of-abstraction phrase for one of the second-level modules (DOCFIG) was: DOCFIG reads the software configuration description, processes operator modifications, verifies memory size, performs central hardware rollcall, performs MPC bootloading, and performs peripheral hardware rollcall. This second-level module was then broken down into six third-level modules (DOGCF, DOMCF, DOMVER, RLLCH, DOMPCB, and RLLPH), which were also defined with level-of-abstraction phrases and the process of top-down design continued.

Top-down design proceeded in this manner until the question was no longer what to do but how to do it. For example: the final what answer for a routine to handle console output was "DISPLAY writes a message to the console, awaits I/O termination, and checks the I/O status." Any further breakdown of this function would involve asking questions such as how does it write to the console?, or how does it await termination? The what phrases implied no logic but simply defined constituent functional components. This process resulted in the successive functional decomposition of the problem with the final level of decomposition defining all modules of the problem solution.

By applying the decomposition procedures

described above to the design of CMU, several problem areas arose. Examples of level-of-abstraction outlines which could be found in existing literature were not explicit enough to answer all of the designers' questions. It was extremely difficult to keep logical decisions out of the process and difficult, at times, to distinguish between what and how answers. For instance, in the example above, one might be tempted to further decompose the function of the DISPLAY module to three individual modules: 1) WRITE writes a message to the console, 2) WAIT awaits I/O termination, and 3) CHECK checks the I/O status. These steps can also be viewed as answers to how DISPLAY works. Difficulties also arose in recognizing when the process of asking what to do had not gone far enough, and in adhering to the goal of minimum decisions at each level-of-abstraction.

Extensive management control had to be exercised during this phase of design to keep designers who were not experienced in structured design techniques from jumping ahead too soon. There was a tendency to want to begin construction before all functions had been fully defined.

While the theorists recommend that each level-of-abstraction be attacked without any attention to details at a lower level, in this application it was impossible to carry out adequate functional decompositions without stopping at many points, suspending decomposition and evaluating what was going to take place in the next few steps. It was impossible to completely eliminate look-ahead.

Despite these problems, this gradual functional decomposition procedure was an extremely positive overall factor in the design phase because it forced a total functional design of the system before any other activity began. By forcing as few decisions as possible at each step of decomposition, the system was indeed broken apart naturally along logical algorithmic boundaries. In almost every case, each module as defined by the lowest level of decomposition did have a single function. Exceptions to this were a few modules which performed several closely related functions unique to that module.

Another useful result of this technique was the design of a functionally related system rather than a procedurally related system. This minimized between-module interface problems.

The second phase in top-down design after writing the levels-of-abstraction phrases is the development

of a Hierarchical Functional Diagram (HFD). The following is a sample portion of the CMU functional diagram:

```
Perform Erase Procedures - Module Number 2                    DOTASK
   Initialization                                             DOINIT
   Configuration Processing                                   DOCFIG
   FNP Initialization                                         DOBOOT
   Clear Processing                                           DOCLER
   Termination                                                DOEXIT
         .
         .
         .
   Configuration Processing - Module Number 2.2               DOCFIG
      Configuration Definition - Module Number 2.2.1          DOGCF
         Read Card Image                                      DORCF
         Process $ MCT Card Image                             DOMCT
         Module Number 2.2.1.1
               .
               .
               .
            Process Memory Size Field                         MSZPR
            Module Number 2.2.1.1.2
               Numeric Field Scan                             SCNUM
               Error Check
               Accumulate Total Memory Size
```

In the above diagram, each line has one of three possible interpretations. If the whole line is underlined, it represents a module expanded in-line as indicated by the indentation of the following lines. If only the entry in the right margin is underlined, the line represents a module expanded elsewhere in the HFD. A line with no underlining represents a lowest-level functionality. The structure of the system is fully defined, including the basic function of each module and its relationships with other modules in the hierarchy. Construction and implementation may now follow this Hierarchical Functional Diagram in a top-down fashion.

The HFD is basically a reorganization and condensation of the previously developed level-of-abstraction phrases. The designer finds all identical and similar functional components and places them at a high enough level in the hierarchy so that they are available for lower functions needing them. For example, the DISPLAY module described earlier would be placed high enough in the hierarchy to be available to all subordinate modules requiring console services. Although not an easy task, this process proved to be extremely beneficial. It enabled the elimination of redundant modules and, with some minor modifications, the redesign of modules with similar

functions, therefore increasing overall system efficiency.

Two additional procedures accomplished during this phase of design were the structuring of an online Support Library and the definition of input and output formats. It is necessary to set up the Support Library at this time so that the documentation can be built in a top-down manner and be available in a useable format for all members of the development team. The complete design of the system is aided by close attention to input and output formats.

1.2 Top-Down Construction

The construction phase of top-down development includes the definition of all control interfaces and data structures plus the explicit specification of the logic for each module. The Program Design Language (PDL), or specification language, is the tool used for this purpose in a Structured Programming application. The PDL can range all the way from a freeform, narrative-type language to a formal language containing precise syntactical and semantic definitions. In both cases, the purpose of the language is to facilitate the translation of functional requirements into computer instructions using the Structured Programming constructs (simple sequence, selection, and repetition).

During the construction phase the control interfaces and data structures are defined before the specification language is constructed.

Specification language generation proceeds in top-down fashion according to the Hierarchical Functional Diagram, answering the question how to do it rather than what to do for each functional component. The answers are written in the specification language, and the result reflects the complete logic of the program. Each functional component becomes a module. In the construction of CMU, a few cases were encountered where an HFD function was represented by more than one module, but, in no instance, was the reverse true.

The specification language should be narrative and precise enough to be definitive. The language chosen for CMU was PASCAL, a formal specification language which forced the designers to explicitly define all structures. Although PASCAL satisfied many of the requirements for a specification language (see "Programming Structures and Conventions" below), there

was unfortunately no suitable compiler available for it on the Honeywell Series 6000. Therefore, the code developed during the construction phase had to be hand-compiled into GMAP. This hand-compilation process, along with system integration and testing, constituted the implementation phase of system development.

The availability of a compiler for the specification language would have resulted in a considerable savings in development resources since the hand-compilation would have no longer been required. Not only would a large portion of the implementor's work have been done by the designers, but only one representation of the logic would have had to be maintained.

During CMU development, it was feasible to define module interfaces before generating the specification code. It was also feasible to lay out data structures in advance but, because of deadline pressures, the development of all control interfaces and data definitions was not completed when construction of the specification code began. This resulted in several problems. Delays in construction due to incomplete definition of interfaces would have been avoided had the interfaces been fully defined prior to commencing construction of the specification language. The design of the internal tables would also have been more efficient, and construction of some of the modules would have been enhanced. The deadline pressures were a result of attempting to meet internal development milestones (i.e., completion of design and construction phases). Had the internal milestones been adjusted such that the recommended procedure could have been followed, it is probable that the final deadlines would not have been adversely affected, and construction problems would have been reduced.

The use of a specification language for laying out program logic was very advantageous. Slight deviations from top-down construction were occasionally required because of time needed to research hardware specifications. This meant that a module in mid-decomposition would be temporarily abandoned, and work would be done on another while the module in question was being researched. A feasibility/research factor considered in the schedule prior to beginning this phase or a project leader thoroughly familiar with the hardware would have precluded most of these deviations; however, no severe problems resulted.

The construction of the specification language

was a very straightforward process because the programmer's attention was concentrated on answering the question how to do it. Having just one function per module, plus the use of a high-level specification language which required less attention to machine details also aided in this process.

One problem encountered was due to the choice of PASCAL as a specification language. Since no Series 6000 PASCAL compiler was available, and since PASCAL user documentation was not always clear, questions on the language semantics were often difficult to answer.

Module "readbacks" (reading the module code aloud to each other) and system "walkthroughs" (reviewing the system step-by-step) are very important aspects of the Structured Programming approach. In theory, each module should be read back to other members of the group, and each group of modules should be walked through to check interfaces and logic. Many of the individual readbacks were bypassed because of the time pressures discussed above. It is likely that some implementation problems would have been avoided if this theory had been applied more strictly. A system walkthrough was performed approximately once a month. This was an essential process and helped identify interface problems, in addition to serving as a medium for design reviews with the customer.

1.3 Top-Down Implementation

The advantages of Structured Programming were felt most strongly during implementation. Implementation is actually made up of three different activities: 1) coding the functions in a machine-recognizable language, 2) integrating the separate modules into a complete system, and 3) testing both the individual modules and the integrated system. The top-down concept of Structured Programming allows all three of these processes to occur simultaneously, greatly simplifying the task and allowing most design discrepancies to be apprehended early.

Implementation again proceeds according to the Hierarchical Functional Diagram. As each module is coded, any modules called by it that have not yet been coded are represented by dummy code (stubs). Integration and testing of a module begin without having to wait for other modules to be coded.

This technique was followed very closely in implementing CMU. Using a top-down approach, each

major function and all of its subfunctions were usually coded, by one person, together as a block of modules. Major functions in this sense were printer I/O and tape I/O.

In order to allow simultaneous testing and generation of code, subordinate modules were initially coded as stubs. Each stub that was written supported the required calling sequence so that when the dummy code was replaced by a complete module, no modifications were required in the calling program. As each module was completed, it was integrated with other completed modules and tested. This completely eliminated the complex and often painful separate integration phase. Because only a small piece of logic was being integrated at a time, the problems were localized and much easier to debug. Problems were caught early enough so that if they involved other modules, changes were easier to coordinate. Since in most cases the implementor was not the designer, yet another person was forced to do a "readback" of the specification code, further increasing reliability.

Because CMU was required to be independent of GCOS, it could not be tested while GCOS was running. In order to most efficiently utilize the dedicated machine time, a debug capability was designed as part of CMU. This capability allowed the real-time insertion of breakpoints, selective dumping of memory, and modification of the program from the console. The use of an online debug routine is not part of the Structured Programming concept; however, the structured implementation of CMU allowed maximum benefit to be obtained from this capability since it was implemented early in the testing phase. For security reasons, the debug capability was removed from CMU before delivery of the program.

Top-down implementation proved to be very efficient. Since all of the algorithms had been designed previously, the programmer needed only to be concerned with the machine representation of the logic, making the resultant code more accurate and efficient.

1.4 Top-Down Maintenance/Modification

In discussing the structured development of a software system, the maintenance and modification phase must not be neglected. Indeed, in this area, Structured Programming techniques present a considerable potential for savings.

Although the Clear Memory Utility has not been extensively modified since its implementation, several maintenance-type changes have been made. The functional organization which resulted from the top-down approach to design made it an extremely simple procedure to locate a problem area and to isolate the modules and data structures affected by the fix. Once a fix was designed, it was thoroughly tested to ensure that no secondary errors were introduced. Any changes made to the implementation code (GMAP) were also made to the specification code (PASCAL). This additional workload of maintaining two representations of the logic would be greatly reduced if the specification language were a compiler language.

In designing a major modification to a software system which was written using a structured philosophy, the modification should also be approached in a top-down fashion. The exact technique employed would depend upon the nature and extent of the particular modification involved. In any event, the structured format of the existing system should be utilized and maintained.

Any documentation which is being maintained in support of the system should immediately be updated. Having that portion of the documentation support library online greatly simplifies this task. Also, due to the structured format of the documentation, it is easy to find the parts affected by the modification.

1.5 Programming Structures and Conventions

PASCAL was chosen as the specification language for the Clear Memory Utility. The most important factor involved in this choice was that CMU had to be "verifiable" for security reasons. Containing virtually no default capabilities, PASCAL requires the coder to specify everything. All data structures are classified as to type and usage and symbolic references are used in all cases. The lack of default capabilities helps reduce the amount of "incorrectness" which is built into the system. In addition, since all aspects of the design are laid out clearly, the design is much easier to verify.

Although strict Structured Programming theory calls for the use of only three logical structures (sequence, selection and repetition), a very limited use of the goto structure greatly enhanced the readability and clarity of the code. Without its use in forward transfer of control out of if-then-else

structures, the logic would have become nested up to ten levels deep, and a great deal of duplicate code would have been required. Used for this purpose and as a mechanism for error transfer to exit procedures, the goto greatly enhanced the coding.

PASCAL's case statement was also used in an effort to reduce the complexity of if-then-else structures. A useful addition to the case statement would have been an all-else capability. The while and repeat statements were used to explicitly define loops. Altogether, over 99% of the program was defined using the recommended Structured Programming structures . It was feasible to use them in a precise, efficient manner.

Each module should be designed with only one entry and one exit point, and control should always be returned to the calling module. The use of these techniques in the development of the Clear Memory Utility resulted in a system which is easy to follow and debug.

One problem encountered in using a specification language was that several of the analysts were primarily assembly-level implementors. Close supervision and monitoring were required during the initial specification work to condition analysts to conceptual-level, rather than machine-level, thinking.

Structured Programming calls for the use of common conventions in the treatment of data. The explicit typing of data in PASCAL allowed all data to be structured to the needs of the algorithms as well as of the hardware. Several functions had to be written to support unique hardware requirements such as pointer handling. However, the data representation was clear and straightforward.

The variant scheme was used extensively to redefine data areas in different ways. In the following example, the packed record MSG can be referenced both as an 80-character array (MSG.V0) or as two arrays, one 7 characters long (MSG.V1F1) and the other 73 characters long (MSG.V1F2).

```
MSG: packed record
   case thrbit of
   0 : (V0: packed array [1..80] of char);
   1 : (V1F1: packed array [1..7] of char;
        V1F2: packed array [8..80] of char);
   end;
```

This is a very efficient method for explicitly

documenting the different ways space is utilized.

Variables were localized where possible. This was made relatively easy because of the modular structure of the program design. Exceptions to this recommended practice were a few variables and system tables which had to be global because of the operating system nature of CMU.

All references to data were symbolic and the data names were kept meaningful - even though up to 20 characters were occasionally required. Use of the PASCAL with statement aided in keeping qualified references in-line. In the following example, IOMtbl[IOMnbr].CHLid[CHLnbr] is used as a qualifier for both the device code (Hypofig.DevVar.DevCod) and number of units (Hypofig.Nunits) in the remainder of the statement:

```
with IOMtbl[IOMnbr].CHLid[CHLnbr] do begin
   Hypofig.DevVar.DevCod := DPTAP;
   Hypofig.Nunits := 2;
   end;
```

Coding was restricted to a maximum of one statement per line, although a few statements required more than one line due to the length of data names and depth of indentation.

Indentation caused some minor problems since a consistent set of rules was difficult to define and enforce. An attempt was made to adhere to internally set standards as much as possible, but an automatic indentation capability would have been the ultimate answer to this problem.

Structured Programming calls for a maximum module length of approximately one page. Due to the complexity of some of the operating system functions, it was not practical to remain within this limit for CMU. Although the mean for the application was 43 lines, 20% of the modules had more than 100 lines and 7% had more than 200 lines. The main advantage of having short modules is that the function of a particular module is sufficiently limited in scope to permit greater clarity. For CMU, the longer modules tended to include case statements with many cases so that the logic was exclusive and parallel. It should also be pointed out that these statistics include not only executable statements but also comments and data-defining statements.

CMU's implementation language was the Series 6000's native assembly language GMAP. Since no PASCAL

compiler for the Series 6000 was available, the implementors hand-compiled from PASCAL to GMAP and then used the GMAP assembler to create object code. A close correlation was maintained between the specification code and implementation code, making debugging extremely easy. As previously mentioned in "Top-Down Construction", the use of a specification language for which a compiler was available would have saved a considerable amount of development resources.

Comments in the GMAP module referred directly to specification statements, and specification code which was repeated more than once (in calling sequences, for example) was always represented the same way during implementation. In the few cases where it was necessary for purposes of efficiency to write implementation code which did not exactly reflect specification statements, comments were included to clearly relate the implementation code to the specification code.

The average specification-to-implementation language ratio was 1 line to 2 lines, an excellent expansion factor.

1.6 Documentation Requirements

Structured Programming theory treats documentation as a tool which is pivotal to the designers and implementors during system development rather than as a necessary evil to be done at development's end for use by maintenance personnel.

For the development of CMU, five separate documents were written. The Work Plan was written prior to the design phase and defined the overall goals of the project, presenting initial time estimates and resource requirements. The Functional Specification, Design Specification, and Test Plan were written during the design, construction, and implementation phases, and the Maintenance Manual was written after the implementation phase had been completed. If scheduling had permitted, the Maintenance Manual would have been much easier to write during implementation rather than as a separate effort. In most instances, the efficient development of CMU was aided by meticulous attention to the concurrent creation of appropriate documentation.

For example, it was natural to develop the necessary functional specifications in parallel with the design effort. Since the purpose of the design is to expose entirely functional levels of abstraction

and hierarchical structure, it was a straightforward process to summarize this information in prose. Once a function was defined, it was immediately documented in detail. Not only was the module's function documented, but also its interfaces with other modules and any constraints which it might impose upon the user. In addition, descriptions of the background and overall objectives were natural products of thought processes which took place at this time. Input/output descriptions were also laid out in the design phase and were included in the functional documentation.

The Design Specification was written during the construction phase of development. Once each module was decomposed to the specification language level and the question how was answered, a design overview was written explaining thoroughly, in prose, the algorithm used to solve the problem. As discussed above (see "Top-Down Construction"), considerable benefit would have resulted from having all the data structures fully defined and documented at this time; however, internal deadline pressures precluded this. The Design Specification included a list showing inter-module dependencies and areas particularly sensitive to change. This work was initiated during the design phase and completed during construction. Documents reflecting the current status of each module were not created. Instead, this information was included in weekly status reports. The specification language representation for each module was kept up-to-date and well-organized for easy reference by team members.

Maintenance documentation was generated in conjunction with the implementation of each module and after the implementation phase was completed. In addition to the Maintenance Manual itself, which thoroughly documents each module and all data structures, extensive comments were coded in both PASCAL and GMAP. These comments correlated the two different representations of CMU. In this way, the specification language form became the basic documentation for the system. At the same time, detailed implementation conventions were documented. With the exception of a very few algorithms, flowcharts were not used for CMU because the specification language clearly showed the logic. If flowcharts had been required, they would have been generated during the construction phase.

Generation of test specifications is also accomplished during development. For this task the test specifications were developed on the overall system in final form near the end of the cycle. Much time would have been saved had module-by-module test

design been accomplished during construction, thereby avoiding the process of reviewing the specification language in detail to obtain this information.

An internal incident control system was implemented during regression testing to report and track problems encountered. The incident report system proved to be a valuable tool which not only documented the problem but provided a mechanism for controlling it through solution, retest, and resolved status.

1.7 Development Support Library

The core of the entire CMU development task was a support library maintained in a form and location convenient to all members of the development team. A Development Support Library (DSL) is defined as a cohesive body of interdependent manual and computer procedures designed to achieve standardized real-time record keeping, documentation, and control for projects in a Structured Programming environment. The DSL, as implemented for the development of CMU, consisted of an online library of current documentation and specification language (maintained through the use of the timesharing text editor) and an offline library composed mainly of copies of all documentation, program listings, and status reports.

Access to both the online and offline libraries was strictly controlled by the programming secretary, who was the only person other than the chief programmer who knew the password to the online files. System designers signed out current listings of modules from the offline library, made changes to the listings, and returned them to the programming secretary, who then applied the changes to the online library. Had additional programming secretary resources been available, it would have been beneficial to maintain the implementation code (over 23,000 lines of GMAP) as part of the online library. However, the implementation code for CMU was maintained in compressed card format by the implementation programmers.

The offline library served as the project archives. Past versions of all documentation, specification code listings, and implementation code listings were maintained in an orderly fashion. All status information was also maintained as part of this library.

The use of this support library produced several very positive effects. Having a historical backup was

priceless while developing the specification language, and having backup copies of documentation was useful for the creation of new documents where sections of the new document could be extracted from old documents. In addition, the use of a central library contributed to group cohesiveness, giving the feeling of a common product.

1.8 The Chief Programmer Team

The Chief Programmer Team (CPT) concept was the basis for the organization of the CMU development task force. As defined in the literature, the CPT consists of a chief programmer, a backup programmer, a programming secretary, and additional support programmers as required. The chief programmer is the architect of the system and holds primary responsibility for the project. The chief programmer should not be assigned coding duties but should be free to design and provide overall control to the structured development process. The backup programmer should participate in all of these functions and serve as a research assistant. The bulk of the coding should be performed by support programmers. The programming secretary enters all code, maintains the support library, and controls all computer runs.

Because of the complexity of the project and because this was the first attempt at Structured Programming for the people involved, the CPT concept was modified for use with CMU. Four people (one of them designated as administrator) served as the nucleus of the project, handling design and other chief/backup functions. The ideal situation would have been for a single person to act as the chief programmer and make all design decisions for the system. There was some vacillation during the early stages of the project in interpreting techniques described in literature, and some time was lost in organizing the Structured Programming approach. Although Structured Programming did make managing the project easier, it should be emphasized that the administrative workload was not eliminated. The heavy technical load precluded proper attention to administrative tasks such as interfacing with management and monitoring system development.

As the project progressed through construction and implementation, people were added to the project as programmers. The organizational structure was a very positive factor here since individual tasks were well-defined and there was little confusion or overlap of duties at this level. A programmer could quickly

become familiar with PASCAL and the implementation conventions and begin implementing the specification code.

The programming secretary held complete responsibility for the organization and maintenance of the support library in addition to the entry, creation, and production of all documents. For a project the size of CMU, these tasks exhausted the resources of one person. As a result, other tasks normally reserved for the programming secretary had to be done by other personnel. All implementation code was keypunched by external support personnel. Programmers were responsible for controlling all computer runs. Additional programming secretary resources would have improved the organization, but were not available. Still, the programmer was successfully kept away from the computer until the testing phase and was not burdened with keypunching or excessive administrative procedures. The only real problem encountered in this area was the quality of the keypunching, and it would have been improved if the programming secretary had done it.

Although additional personnel would have been required for strict adherence to the CPT concept, it probably would not have been cost-effective to hire them.

2.0 MANAGEMENT

2.1 Planning

Many of the difficulties encountered during CMU development can be traced directly to two conditions which arose early in the course of the task. First, the resource requirements initially were seriously underestimated. Secondly, module construction was scheduled to be done concurrent with the Design Specification development.

Before the resource requirement problem was resolved, the tasking statement was amended twice, the calendar time allotted to the task more than doubled, and the cost more than tripled the original estimate.

The original estimate was based on a qualitative evaluation of task complexity, size of intended user community, programming languages, learning curves, etc. Much of the resource estimation error was due to the assumption that I/O and Configuration Description/Verification could be easily adapted from GCOS Startup. The tasking statement specifically required that 1) the program "be verifiable as to correctness", and 2) "procedures for this task will follow a Structured Programming approach". GCOS Startup, in contrast, was designed with goals of memory and execution time efficiencies. Thus adaptation from GCOS Startup would have introduced a large component of unstructured code into CMU.

At the time of the first task amendment, two months into the task, the I/O portion of the task was still in its early stages, and the full magnitude of the resource estimation problem was not yet apparent. In retrospect, the I/O portion of the task was on the critical path. Its design should have begun earlier, and more personnel should have been initially assigned. Downstream, the learning curve was such that it became difficult to add a second I/O designer.

The first task amendment added seven calendar weeks to the task, but still relied on adaptation from GCOS Startup. When the first amendment did not solve the problem, the contractor undertook a major re-evaluation of the task. The re-evaluation took place four months into the task, in parallel with completion of the Design Specification. The second task amendment grew out of that re-evaluation, and all delivery milestones from that amendment have been met. At the time of the second task amendment, a conscious decision was made to follow a fully-structured

approach and write all I/O and configuration processing logic from scratch. Also, the writing of the Maintenance Manual was rescheduled to commence after the delivery of the software.

The re-evaluation which led to the second task amendment suggests, in part, what can be done to obtain more accurate estimates on future projects. Instead of attempting to estimate resource requirements for the entire project before a detailed analysis has taken place, first a Functional Specification should be tasked separately. Only when that task is done should contractual estimates be made for Design Specification resources. Similarly, no contractual estimates should be made for the Implementation task until the Design Specification task is complete.

The second condition which contributed to development problems, module construction before Design Specification completion, was in part an outgrowth of the resource estimation problem. The calendar time allotted to the task necessarily forced some overlap. It was felt that the design and construction could proceed in parallel, coordinated such that design of a major program block would be completed before construction of that block began.

The progress of the task suffered because of the overlap. The analysts were forced to switch repeatedly between design logic and construction logic and, consequently, were less efficient at each. The data structures were not completely defined before construction. Required changes to some data structures took longer because of the need to modify procedural code affected by the changes. Other data structures could have been restructured for more efficient usage, but were not because it would have required changing too much of the existing procedural code.

At the very least, the I/O design should have been scheduled earlier than it was. Ideally, the Design Specification should have been delivered before construction began.

2.2 Personnel

As described above ("The Chief Programmer Team"), CMU was staffed using a variation of the CPT concept. A design team was used instead of one chief and one backup programmer, and some functions which, in theory, should have been performed by the programming secretary were delegated to programmers and overhead

support services. More calendar time and an additional part-time programming secretary would have been required to strictly apply the CPT concept. Had the application been strict, the design would have been more homogeneous, and addition of construction/implementation programmers would have been deferred.

Structured Programming theory includes "egoless programming" as a benefit. For CMU, the benefit was neither complete nor easily attained, but there was indeed a benefit. A CPT organization requires people who are receptive to team loyalty and/or can be motivated to displace their subjective reactions. While the CMU team was by no means composed of "primma donnas", the designers were senior individuals, many of whom were accustomed to sole custody of a programming task. Motivation for team loyalty was provided at first by orientation training in Structured Programming concepts. This motivation was reinforced by peer pressure and by the visibility of an individual's work during module readbacks and system walkthroughs.

What emerged during the task was a group ego in addition to the individual egos. Group morale was low during the re-evaluation which preceded the second task amendment, but improved and remained high as milestones thereafter were successfully met.

If the CPT concept had been strictly applied, the magnitude of the individualistic ego problem would probably have been less at the beginning. The team loyalty benefit would also have been enhanced by additional module readbacks which had been planned but were bypassed because of deadlines. The design reviews (system walkthroughs) with the customer were immensely beneficial, as the customer was included in the process instead of becoming an adversary.

Although the operating system nature of CMU required its developers to be senior programmers, no above average talent was required because of the use of Structured Programming. In staffing a project, however, a manager should not assume that Structured Programming is a replacement for qualified personnel. The development team members should be familiar with Structured Programming theory and techniques and, ideally, have had experience on other projects using Structured Programming.

Training was a vital part of the CMU success, benefitting directly from close adherence to Structured Programming conventions. Mixed success was

observed in the addition of programmers to the team in midstream. Because of the documentation produced, early additions to the team were quickly trained and soon contributing to the task. Later, because the second amendment to the task delayed the production of the Maintenance Manual, documentation lagged behind the design, and training became more difficult. In this respect, it would have been better if the task amendment had still required that the documentation be kept concurrent.

CMU was a complex system, and the learning curve was such that it was not cost-effective to add a programmer on a temporary basis shorter than two months.

2.3 Control

Productivity and morale are enhanced by increased visibility of the program as it progresses toward completion. Structured Programming theory incorporates module readbacks, system walkthroughs, and the Development Support Library to boost visibility.

For CMU, the number of module readbacks was reduced because of time pressures, but those which did occur helped promote dialogue. More time should have been allotted so that every module could have been subjected to a readback. The authors of this document, for example, have performed "module readbacks" as each section of the document has been outlined and drafted.

System walkthroughs took the form of design reviews with customer participation. Had time permitted, internal system walkthroughs would also have been desirable. The design reviews were scheduled at an average frequency of once per month and concentrated primarily on the design and construction phases of the task. The customer feedback provided invaluable and timely insights to both customer and contractor. The customer became progressively more knowledgeable about the consequences of each system requirement and was able to meaningfully discuss technical and aesthetic considerations not explicitly provided by the tasking statement. The contractor was able to implement many of these considerations or negotiate alternatives based on first-hand knowledge of the customer needs. Neither customer nor contractor should be expected to read the other's mind. There is no substitute comparable to the communication provided by design reviews, and their scheduling is strongly encouraged for future projects.

As predicted in the literature, the use of the Development Support Library also enhanced program visibility. Team members may fall ill or take vacation during a task. There must be written as well as verbal communication during normal project periods. Concerning program visibility, the main benefit derived from the library was the elimination of the question of where to look for up-to-date documentation.

CMU management control also included Work Plan milestone charts, internal weekly status reports, external monthly status reports, the Demonstration Test Plan, an incident reporting system, development team meetings, and in-process reviews.

The Work Plan milestone charts provided a gauge against which progress could be measured. Unfortunately, until the second task amendment, they were also a source of frustration. Once the resource estimation problem was corrected, internal milestone charts expanded the Work Plan charts and provided an excellent control for resource allocation and cost-effective decisions. For example, the Program Maintenance Manual could not have been delivered on time if everything desired by the analysts had been included. The potential problem was recognized early enough so that the scope of the document could be made cost-effective while optimizing the quality within that constraint.

Of course, the milestone chart gauges would have been useless without monitoring. The internal and external status reports kept management and the customer formally apprised of progress and problems on a timely basis. The status reports are included in the Development Support Library archives. They greatly aided the development of this document, and they serve as a guide to effective management of future Structured Programming tasks.

The Demonstration Test Plan and the incident reporting system provided increased confidence in the accuracy of the program. Both, however, should have been formulated earlier than they were. The Demonstration Test Plan was written after the Design Specification but was based more on the Functional Specification and on the code which had been constructed up to that point. If the module descriptions in the Design Specification had included the testing requirements for each module, the Demonstration Test Plan would have been easier to assemble and would have provided a more rigorous test. Similarly, the experience with the incident reporting

system indicates that it should have been instituted, at the latest, with the first implementation code assembly. Even more benefit would have been derived if it had been established during construction and/or design. Unresolved incidents were summarized in the internal status reports. Thus exception path incidents were not allowed to be overlooked.

Development team meetings and in-process reviews were held on an as-needed basis and aided communication internally as well as with customer management. Human nature being what it is, the internal meetings occasionally served to unveil accumulated grievances separate from the meeting agenda. On a large project such as CMU these internal meetings should be formally scheduled weekly or at least bi-weekly. In-process reviews should also be formally scheduled - once per quarter and/or once per major milestone is suggested.

3.0 BENEFITS/COSTS

3.1 Productivity

One measure of productivity is the amount of time spent on each phase of the project. Over a 13-month time period, six people worked on CMU. Time spent in different phases is summarized as follows:

A. Design	
1. Hierarchical Functional Diagram (HFD)	- 14MM
2. Specification Language (PASCAL)	- 21MM
	35MM
B. Implementation and Testing	
1. Implementation and Integration Testing	- 14MM
2. Regression Testing	- 2.5MM
	16.5MM
C. Training	- 7.5MM
	7.5MM
D. Programming Secretary (Documentation)	
1. Design Specification	- 2MM
2. Test Plan	- 1MM
3. PASCAL Code	- 4MM
4. User's Guide	- 1MM
5. Maintenance Manual	- 2MM
	10MM
Total	- 69MM

The above figures show that approximately 51% of the man-months for the entire project was spent in the design phase. If training and documentation activities were excluded from this calculation, the design phase would account for a full 68% of the total. If a PASCAL compiler had been used to generate the GMAP instead of the somewhat tedious process of hand-compiling, the implementation time would have been reduced drastically, and a substantial improvement of these already impressive statistics would have resulted.

This emphasis on design resulted in a very fast and efficient implementation. This would have been even more evident had the chief programmer had more time to design and review code. In addition, it must be emphasized that this was the first experience with structured system development for the design team, and the learning overhead will be greatly reduced in future projects.

A second measure of productivity is the amount of code produced during each Structured Programming

phase, summarized as follows:

A. Hierarchical Functional Diagram (HFD) - 1000 lines
B. Specification Code (PASCAL)
   1. 205 modules
   2. 10,837 executable statements
   3. 43 executable statements/module average

C. Implementation Code (GMAP)
   1. 23,623 lines
   2. 70 lines/day/person of debugged GMAP - counting only implementation and test time. (22 lines/day/person if design time also included).

D. Documentation - 1,933 pages
   1. Design Specification - 532 pages
   2. Test Plan - 162 pages
   3. PASCAL Code - 500 pages
   4. User's Guide - 192 pages
   5. Maintenance Manual - 547 pages

Note: The Functional Specification is not included in the list above because it was produced before the programming secretary was hired.

The above figures reflect a 1-to-2 expansion ratio from PASCAL to GMAP. Because of a task amendment, the Maintenance Manual was produced following delivery of the software. Otherwise, all of the documentation listed above was written by the programmers as a natural byproduct of the design and implementation processes. Had this been done "after the fact," the cost would have been considerably higher.

3.2 Reliability

An important measure of the effectiveness of a development scheme is the number of errors found in the system which is developed. The following statistics were accumulated in an attempt to add this perspective to the productivity figures.

A. Functional Design - only two significant errors were discovered which involved modifications to the functional design (HFD). For example, the MPC bootloading was initially designed to be done after peripheral hardware rollcall. Because of security requirements, the sequence of events had to be reversed.

B. Detailed Design - only five significant design errors were encountered which involved modifications to the specification language (PASCAL). For example, the original design for processing of the FNP clear object deck did not allow for its 18-bit word representation, and some redesign was required.

C. Implementation Code (GMAP) - an average of two errors per module were found (primarily keypunch errors which were quickly caught).

D. Regression Testing - 39 errors, mainly minor implementation errors such as misspellings in error messages, were uncovered.

E. Customer post delivery Q/A - only nine errors were reported. These errors were all minor implementation errors which had not been caught in predelivery testing and were primarily in the areas of configuration card and clear command editing.

These figures reflect a very high degree of reliability, proving that the rather impressive production figures are not discredited by an excessive error rate. The small number of errors is attributed to the precise design and implementation techniques dictated by Structured Programming.

3.3 Training

The fact that there was a moderate overhead in training costs, especially in the early phases of development, is attributable to lack of familiarity with Structured Programming. In addition to this, there was a certain amount of formal training to familiarize team members with the developmental approach and the PASCAL language. For the initial CMU task force this accounted for approximately five man-months of time. During the development of CMU, three new people joined the team and had to be brought up to speed. This proved to be a very easy task for several reasons:

1) The highly organized and logical breakdown of the project,
2) Availability of documentation,
3) Explicit functional design, and
4) Explicit construction of specification language.

3.4 Maintainability

Structured Programming enhanced the maintainability of CMU. The specification language served as a guide to the interpretation of the implementation language and an overall view of the capabilities and relationships was readily obtainable from the HFD. The team members found it very easy to understand and follow one another's modules.

Because of the ease of understanding, modifications to the system were usually straightforward, and their impact upon other modules was easy to determine.

3.5 Management

In developing CMU, skilled personnel were still required, despite the use of Structured Programming techniques. The CMU development team was composed of senior people with an in-depth understanding of hardware/software interfaces and operating system design problems. Given sufficient time and a true chief programmer, the task would not have required so many senior people.

Structured Programming did not eliminate the administrative burden of the project. However, because the organization was so formal and individual tasks so well-defined, the burden of day-to-day direction of individuals was eased.

It was also found that communications between individuals was much easier, making it a straightforward job to evaluate the progress of the task. Initially, estimating the time requirements proved to be a difficult problem. Detailed knowledge of project requirements was still required before accurate time estimates could be made. To the extent that Structured Programming aided in defining what had to be done, it helped in the estimation process, but, prior to the completion of the Design Specification, it was difficult to gauge completion time figures accurately.

A thorough design should have been completed before total time-to-complete project estimates were made. All estimates made after the design was completed were very accurate because few unforeseen problems were encountered. This is attributable to some extent to the use of Structured Programming. Management must understand that when considering a Structured Programming approach, the design must be

completed in sufficient detail to identify potential problem areas before "hard" milestones are set. This is not really unique to Structured Programming because even in traditional development environments accurate milestones are much easier to set after "all" potential problem areas have been identified. The main benefit of Structured Programming to the planning function is that the organized and disciplined approach to design makes it easier to identify problem areas early in the development cycle before implementation is begun.

3.6 Other

The consistency which comes from following Structured Programming standards was beneficial in several ways. Adhering to coding conventions made it easy for the team members to understand each other's code. The use of documentation standards and a programming secretary to create the documents online resulted in the production of high quality, useful documents.

The organization of the Chief Programmer Team allowed each individual to have a specialized task. Team members were not required to function in multiple capacities. Senior talent was directed at complex design efforts; less experienced programmers undertook implementation; and the programming secretary entered documentation and maintained the support library. Having various members perform in their specialties resulted in high quality, efficient output at all levels.

Another positive characteristic of the CMU development was that all aspects of the system were more visible to the project members. The support library made each module immediately available to everyone, aiding communications tremendously. All aspects of the design were laid out in a readable format. System walkthroughs and program readbacks resulted in more than one person being familiar with each functionality.

4.0 CONCLUSIONS

4.1 Negative

1. Time pressure forced fewer readbacks than desired. Allocating sufficient time for a detailed readback of each module would have further reduced the error rate and also the number of design modifications made during implementation.

2. Additional design support would have freed the chief programmer and backup programmer to review code and perform other control duties.

3. Nonavailability of a PASCAL compiler resulted in a significant increase in implementation time. Certain PASCAL semantic problems were unanswerable and had to be solved by making an interpretation and enforcing it. PASCAL was not ideally suited for an operating system application because certain hardware-specific functions could not be properly represented.

4. It was, at times, difficult to interpret the definitions of Structured Programming techniques and structures found in the literature.

5. An additional part-time programming secretary would have enabled online entry of source code and taken away from the programmers the workload of running assemblies. The programming secretary could not have accomplished the testing for CMU as suggested by the literature because a considerable amount of debugging was done online while testing. The online nature of the debugging was not a requirement of Structured Programming but rather of the operating system nature of CMU.

6. The initial time and resource estimates for CMU development were inaccurate because they were calculated before the functional design was completed.

7. Time constraints forced the construction phase to begin before the functional design was completed. This should not have been done.

8. Due to the inexperience of some development team members in Structured Programming applications, a certain amount of vacillation occurred (especially during the early phases of the project) in applying Structured Programming theory.

4.2 Positive

1. Productivity was increased because no separate testing phase was required. All testing was done parallel to implementation.

2. Emphasis on design and common conventions resulted in extremely reliable code with relatively few errors detected during implementation and testing.

3. Documentation quality was enhanced.

4. Maintenance and modification of CMU was facilitated due to the structured format of the design, code, and documentation.

5.0 RECOMMENDATIONS

5.1 Organizational/Managerial Recommendations

Program managers should consider it imperative to seek a task definition which is a compromise between 1) making requirements specific to aid top-down design, and 2) keeping the requirements flexible enough to allow cost-effectiveness tradeoffs and program evolution. It is much easier and more cost-effective to analyze and compromise before signing the contract than to amend a task downstream.

Therefore, one of a manager's first objectives should be to determine the complexity of the task. A quantitative complexity assessment can then be used to determine whether, and how, a task is to be subdivided and which documents will be required. To illustrate a method for complexity assessment, the following chart is adapted from a Department of Defense publication on documentation standards (see (1) below):

(1) DOD Manual 4120.17-M, Automated Data System Documentation Standards Manual, December 1972.

WEIGHTING FACTOR

Factors	1	2	3	4	5
1. Originality Required	None (Reprogram on New Equipment)	Minimum (Stricter Requirements)	Limited (New Interfaces)	Considerable (Existing State-of-Art)	Extensive (Advance State-of-Art)
2. Degree of Generality	Highly Restricted (Single Purpose)	Restricted (Parameterized for a range of capacities)	Limited (Some format flexibility)	Multipurpose (Range of subjects)	Very Flexible (broad range of subject matter on different equipment)
3. Span of Operation	Local or Utility	Component Command	Single Command	Multi-Command	DOD Worldwide
4. Change in Scope and Objective	None	Infrequent	Occasional	Frequent	Continuous
5. Equipment Complexity	Single Machine Standard Peripherals	Single Machine Extended Peripherals	Multi-Computer Standard Peripherals	Multi-Computer Extended Peripherals	Master Control System
6. Personnel Assigned	1 - 2	3 - 5	5 - 10	10 - 18	$\geq$ 18
7. Development Cost	Under 15K	15K - 75K	75K - 300K	300K - 750K	> 750K
8. Criticality	ADP	Routine Operations	Personnel Safety	Unit Survival	National Defense
9. Average Required Response to Program Changes	$\geq$ 2 wks	1 - 2 wks	3 - 7 days	1 - 3 days	< 24 hrs
10. Average Required Response Time to Data Inputs	> 2 wks	1 - 2 wks	1 - 7 days	1 - 24 hrs	$\leq$ 1 hr
11. Programming Language(s)	High-Level	High-Level + limited assembly	High-Level + extensive assembly	Assembly	Machine
12. Concurrent Software Development	None	Limited	Moderate	Extensive	Exhaustive
Column Totals	3x1	3x2	1x3	4x4	1x5
33	3	6	3	16	5

The underlined items and the numbers entered at the bottom of the chart illustrate how the chart could have been used for CMU. The next chart indicates suggested subtasking and documents based on total system complexity:

Complexity	Documents (1)	Subtasks
12 - 16	UM	unified
14 - 26	UM OM MM DT	unified
24 - 38	FS DS DT UM OM MM	design implementation
36 - 50	FS DS DT UM OM MM RT	formulation design implementation
48 - 60	FS DS DT UM OM MM RT	formulation design implementation quality assurance

Where separate subtasks are indicated by the above chart, each subtask should be completed before the succeeding subtask is begun. Ideally, estimates should be made and contracts signed for a given subtask only when the preceding subtask is complete.

Naturally, specific task requirements may necessitate changes in the above scheme. For example, the calculated complexity value of 33 for CMU should probably be raised because of its independence from GCOS and the requirement that it be verifiable.

(1) DS = Design Specification
DT = Demonstration Test Plan
FS = Functional Specification
MM = Maintenance Manual
OM = Operator Manual
RT = Results of Testing
UM = User Manual

Another item which must be considered before signing of the contract is the scheduling of milestones. For a contract of more than approximately 3 months duration, there should not be a 100% personnel commitment in order to meet a calendar deadline. To allow for training, vacation, illness, etc., a figure of 80-90% is a realistic maximum, depending on task duration and number of persons assigned. The contract milestones must relate to the various Structured Programming phases. Where indicated, time for research and/or training must be factored into the milestone schedule. Learning curves are additional factors. Sufficient time must be allowed to enable production of Functional Specification, Design Specification, and implementation - all in proper sequence.

Contractor-customer communication should also be formalized before the contract is signed. A design review should be scheduled at least once a month, possibly twice a month if more than two designers are assigned. An in-process review should be scheduled once per quarter, and written status reports should be sent to the customer once a month.

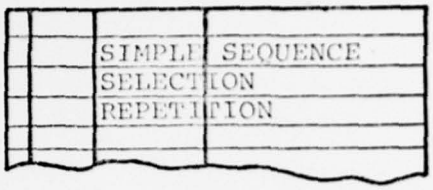
If four or more persons are assigned to the task, a CPT organization is suggested. Otherwise, an individual or a development team may be assigned to the task. For most tasks, the chief programmer will also serve as task leader. In a very large task, task leadership may be assigned to the backup programmer or to another individual. A programming secretary should be assigned immediately to process design documentation. Addition of programmers should be deferred until the implementation phase of the task.

Immediately after signing of the contract, the task leader should develop a Work Plan for the Functional Specification, require weekly status reports from all task members, schedule weekly task member meetings, and institute an incident reporting system. As the system is designed, the Work Plan milestones for succeeding task phases should be expanded through a decomposition process until internal milestone charts are developed to monitor task progress on a weekly basis.

Once the initial task planning has been completed, the task leader must ensure that necessary training is provided, documentation is up-to-date, module readbacks and system walkthroughs are performed, and customer-contractor communication is maintained. These task leader responsibilities are independent of task size.

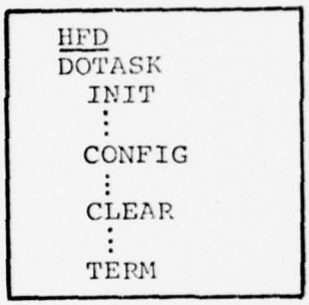
It is not necessary to implement all aspects of Structured Programming in order to derive some benefit. However, there is a recommended hierarchy for implementation of particular techniques which should be followed. Figure 5.1.1 represents this hierarchy and illustrates the interdependencies which exist.

For example, it would be counterproductive to start with a chief programmer but not use the structured coding techniques. However, benefit can be obtained from structured coding techniques and top-down design even in the absence of a programming secretary and chief programmer team organization.



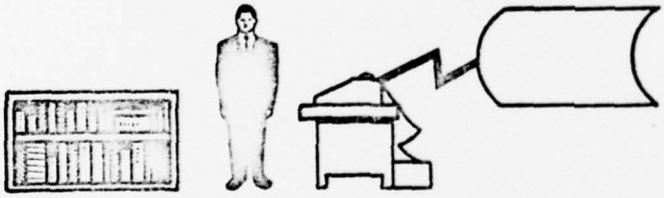
LEVEL 1

- STRUCTURED CODING
- CONVENTIONS



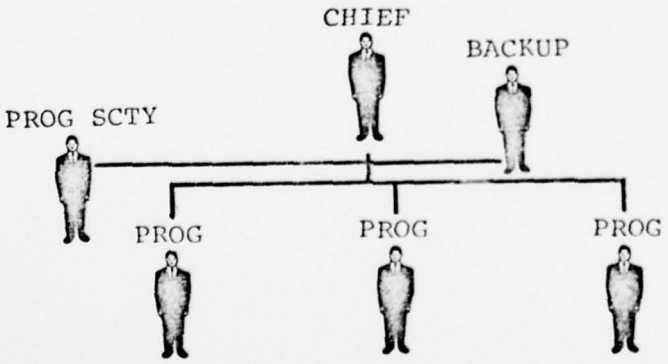
LEVEL 2

- LEVEL 1
- TOP DOWN DESIGN



LEVEL 3

- LEVELS 1, 2
- PROGRAMMING SCTY
- DEVELOPMENT SUPPORT LIBRARY



LEVEL 4

- LEVELS 1, 2, 3
- CHIEF PROGRAMMER TEAM
- TOP DOWN CONSTRUCTION
- TOP DOWN IMPLEMENTATION

FIG. 5.1.1

RECOMMENDED LEVELS OF STRUCTURED PROGRAMMING APPLICATION

5.2 Design Recommendations

Top-down design should be used wherever possible. Ideal candidates are tasks or subtasks which require development of a complete program or a complete module. If a task involves a series of minor changes to a nonstructured system, traditional methods may prove more cost-effective. Between the two extremes, a task may be internally subdivided so that top-down design is applied to as many of the task elements as possible. The fact that a task requires modification of an existing system does not necessarily mean that Structured Programming must be abandoned.

The exact form of top-down design for a particular task depends on task complexity, discussed above under "Organizational/Managerial Recommendations". The design should progress in sequence from levels-of-abstraction, to HFD, to Functional Specification, to Design Specification. The Design Specification should identify the data structures, module interfaces, module logic, and test criteria for each module. If required for the task, a Demonstration Test Plan should be developed from the Functional Specification and the test criteria within the Design Specification.

One final design recommendation concerns data structures. Dynamic (execution-time) storage allocation is more easily maintained and modified. The tradeoff is a higher memory and run-time requirement.

5.3 Implementation Recommendations

The top-down approach should be followed for implementation as for design. Module readbacks and system walkthroughs are vital; even if a task has only one person assigned, that person should seek out the manager, the manager's delegate, or a peer for consultation.

Module stubs should be coded, as required, to enable integrated testing. Unit testing may be required for operating system modifications, but testing of each such unit should be integrated.

The implementation phase is also the time when the Maintenance Manual should be written. It should be a descendant of the Design Specification, and should be developed parallel to the implementation code. If changes are made to functionality or design, the corresponding documents must be modified.

5.4 Coding Recommendations

To maximize the efficiency of the Structured Programming approach, the implementation language must support the Structured Programming logical structures. A compiler must be available. For example, CMU could have been implemented to a great extent in PL/I rather than GMAP. Such an implementation would have sacrificed little of the precision of PASCAL, but would have reduced both implementation effort and maintenance effort.

Once the implementation language has been chosen, the use of the logical structures must be enforced. The goto statement is to be avoided except as specifically justified on a case-by-case basis. When used, the goto must be restricted to a forward jump within a module.

Symbolically-referenced data structures are highly recommended for ease of maintenance and modification. The use of variants to explicitly define different uses of a record is particularly helpful.

Debugging is immensely aided by the convention of narrow module interfaces. A module can be easily understood with one entry and one exit. Module length may vary slightly depending upon the type of application. The general goal should be to restrict module size to approximately one page or 50-75 lines. It should be emphasized that this is a general goal and may be impractical for some modules.

Also recommended are the conventions of indented code and a maximum of one statement per line. Comments should be used liberally, both line-by-line and in paragraph form to enhance the communication provided by the symbolically-referenced data.

5.5 Documentation Recommendations

The documentation produced should be commensurate with the complexity of the task (see "Organizational/Managerial Recommendations" above). Each document should be developed concurrently with the appropriate task phase - for example, the Maintenance Manual should be written parallel to top-down implementation.

For ease of modification, the documentation should be maintained online. The quality is optimized by assignment of a programming secretary to format, enter, update, and produce the documentation. A

history file or archives should be maintained offline, primarily to provide online file error recovery, and also to provide a record of good and bad decisions as a guide for future tasks.

5.6 Maintenance Recommendations

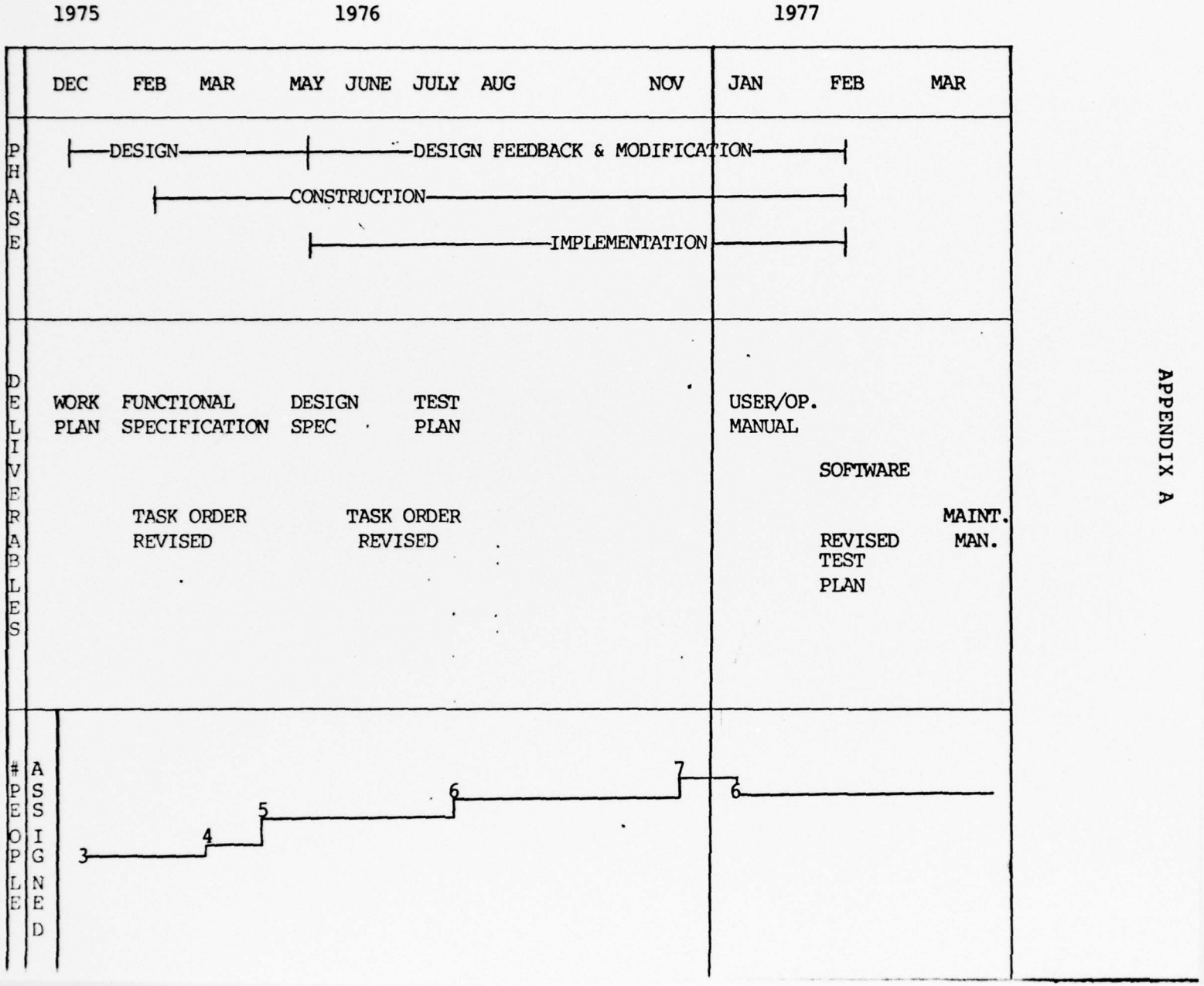
Patches should be made only for emergencies. When a change is made to the implementation code, the corresponding change should also be made to the Maintenance Manual, Design Specification, and/or Functional Specification. In the case of CMU, the construction (PASCAL) code would also require updating. Online documentation eases the updating burden.

A means should be provided for coordinated dissemination of program and/or document changes. Obsolete documentation should be explicitly identified and removed from circulation.

A new programmer must be trained before being permitted to modify a program. To modify CMU, a programmer must be conversant with the documentation, the PASCAL code, and the GMAP code. Module readbacks, system walkthroughs, and testing should be considered as important for maintenance as for development. They may, in fact, be more important because of the newcomer's relative inexperience with the particular system. In general, the Structured Programming techniques used to develop a system should continue to be applied during the system's maintenance.

SUMMARY

There are certain "universal" software development techniques which both Structured Programming and traditional development methods have in common. While CMU definitely benefitted from: 1) disciplined organization, 2) modular code, and 3) coding conventions; a traditional environment would also have derived benefit from such techniques. The success in this development was due not only to the use of these methods but, to a great extent, to the use of other techniques unique to Structured Programming - the Chief Programmer Team and a top-down approach to design, construction, and implementation.


1975
1976
1977
DEC
FEB
MAR
MAY
JUNE
JULY
AUG
NOV
JAN
FEB
MAR
PHASE
DESIGN
DESIGN FEEDBACK & MODIFICATION
CONSTRUCTION
IMPLEMENTATION
DELIVERABLES
WORK PLAN
FUNCTIONAL SPECIFICATION
DESIGN SPEC
TEST PLAN
USER/OP. MANUAL
SOFTWARE
TASK ORDER REVISED
TASK ORDER REVISED
REVISED TEST PLAN
MAINT. MAN.
PEOPLE ASSIGNED
3
4
5
6
7
6

BIBLIOGRAPHY

1. Aron, J. D., Estimating Resources for Large Programming Systems, FSC 69-5013 Federal Systems Center IBM, Gaithersburg, Md., 1969, pp 1-21, cited in RADC Structured Programming Series Volume II, January 1975.

2. Baker, F. Terry, "Chief Programmer Team Management of Production Programming," IBM Systems Journal, November 1, 1972.

3. Baker, F. Terry and Mills, Harlan D., "Chief Programmer Teams," Datamation, December, 1973.

4. Boehm, Barry W., "Software Engineering," IEEE Transactions on Computers, December, 1976.

5. Böhm, C. and Jacopini, G., "Flow Diagrams, Turing Machines and Languages With Only Two Formation Rules," Communications of the ACM, May 1966, pp 366-371.

6. Brooks, Frederick P. Jr., The Mythical Man-Month Essays on Software Engineering, Addison-Wesley, 1975.

7. Canning, Richard S., "The Advent of Structured Programming," EDP Analyzer, June, 1974.

8. Dahl, O. J. et al, Structured Programming, Academic Press, 1972.

9. Dijkstra, E. W., "GOTO Statement Considered Harmful," Communications of the ACM, March, 1968.

10. Donaldson, James R., "Structured Programming," Datamation, December, 1973.

11. Honeywell Information Systems, Structured Programming Guidelines, Vols 1 & 2, Aerospace Division, St. Petersburg, Florida, July 1976.

12. Honeywell Information Systems, Systems Development Library, Federal Systems Operations internal paper, McLean, Virginia, 1976.

13. McCracken, Daniel D., "Revolutions in Programming, an Overview," December, 1973.

14. Mills, H. D., "Top-Down Programming in Large Systems," Current Computer Sciences Symposium 1, New York University, Randall Rustin (Ed.), Prentice-Hall, June 1971, pp 41-55.

15. Peters, Lawrence, "Managing the Transition to Structured Programming," Datamation, May, 1975.

16. Yourdon, Edward, Techniques of Program Structure and Design, Prentice-Hall, 1975.

Unclassified

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

REPORT DOCUMENTATION PAGE		READ INSTRUCTIONS BEFORE COMPLETING FORM
1. REPORT NUMBER TR121-77	2. GOVT ACCESSION NO.	3. RECIPIENT'S CATALOG NUMBER
4. TITLE (and Subtitle) WWMCCS ADP STANDARD CLEAR MEMORY/ MAGNETIC STORAGE UTILITY STRUCTURED PROGRAMMING EVALUATION TECHNICAL REPORT TASK 621, SUBTASK 2		5. TYPE OF REPORT & PERIOD COVERED FINAL 3/29/77 - 6/20/77
		6. PERFORMING ORG. REPORT NUMBER
7. AUTHOR(s) Jeff Karas Robert Hickey Don MacKellar		8. CONTRACT OR GRANT NUMBER(s) Contract DCA100-73-C-0055
9. PERFORMING ORGANIZATION NAME AND ADDRESS Honeywell Information Systems 7900 Westpark Drive McLean, Virginia 22101		10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS
11. CONTROLLING OFFICE NAME AND ADDRESS Defense Communications Agency Command & Control Technical Center WWMCCS ADP Directorate Reston, Va. 22090		12. REPORT DATE June 20, 1977
		13. NUMBER OF PAGES 44
14. MONITORING AGENCY NAME & ADDRESS (if different from Controlling Office)		15. SECURITY CLASS. (of this report) Unclassified
		15a. DECLASSIFICATION/DOWNGRADING SCHEDULE

16. DISTRIBUTION STATEMENT (of this Report)

17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from Report)

18. SUPPLEMENTARY NOTES

19. KEY WORDS (Continue on reverse side if necessary and identify by block number)

Structured Programming
Operating Systems
Productivity Results
Chief Programmer Team
Top-down Design
WWMCCS Software

20. ABSTRACT (Continue on reverse side if necessary and identify by block number)

The Clear Memory Utility (CMU) Structured Programming Evaluation Technical Report examines Structured Programming development techniques used during the development of a stand-alone, "mini-operating system" utility which clears all magnetic storage of the Honeywell Series 6000 computer between periods of classified processing. The effects of the use of the Chief Programmer Team and top-down design, construction and implementation on productivity, reliability and verifiability are examined.

DD FORM 1 JAN 73 1473 EDITION OF 1 NOV 65 IS OBSOLETE

Unclassified

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)